CHAPTER 13

Authorizations and Sessions

Authorizations and *sessions* are among the most important concepts in TPM 2.0. Authorizations control access to entities in the TPM, providing many of the security guarantees of the TPM. *Sessions* are the vehicle for *authorizations* and maintain state between subsequent commands; additionally, *sessions* configure some per-command attributes such as encryption and decryption of command and response parameters and auditing. This chapter describes sessions as they relate to authorization of actions on entities. Chapters 16 and 17 describe details of the per-command session use modifiers.

Authorizations and sessions represent a large topic, so this chapter will proceed as follows:

1. You'll learn some new terms specific to sessions and authorizations. You are advised to review the definitions in Chapter 5 as well.
2. You'll see password, HMAC, and policy authorizations at a high level, along with the security properties of each.
3. The chapter clarifies the differences and commonalities between sessions and authorizations, as well as some aspects of the specification that can be confusing.
4. You'll drill down into some aspects of authorizations that apply to all three types of authorizations: password, HMAC, and policy. You will learn about the authorization roles and the authorization area in the command and response byte streams.
5. You will examine the different types of authorizations in detail, from simplest to most complex: password, HMAC, and policy. After looking at password authorizations, you will see some common aspects of HMAC and policy authorizations, followed by the details of HMAC and policy authorizations.
6. Finally, all the authorization types are tied together into a combined authorization lifecycle.

This chapter doesn't describe the various policy authorization commands. Nor does it describe decrypt, encrypt, and audit sessions, other than to note that sessions are the vehicle for setting these.

This chapter uses diagrams, logical flows, and working code examples to illustrate how authorizations and sessions work. This material is foundational to understanding TPM 2.0. Get ready for a deep but rewarding dive.

Session-Related Definitions

Before you delve into this subject, you need to clearly understand some new terms. These are in addition to the terms described in Chapter 5; you should refer to those definitions as well as these while reading this chapter:

- *Session creation variations:* These are set at session creation time and last for the lifetime of the session. They determine how the session and HMAC keys are created and how the HMAC is generated. There are two choices here: bound vs. unbound, and salted vs. unsalted. The combination of these two choices results in four session variations. These are discussed in detail later. For now, here are high-level descriptions:
    - Bound sessions essentially "bind" the authorization to some entity's authorization value. This binding is done by including the bind entity's authorization value in the session key generation. This affects all calculations that depend on the session key, including HMAC, policy, encryption, and decryption calculations.
    - An unbound session doesn't use a bind entity's authorization in the session key generation.
    - A salted session adds extra entropy, the *salt*, into the session key generation; similar to bound sessions, this affects all calculations that depend on the session key. The extra entropy is sent to the TPM in encrypted form, the encrypted salt parameter which is passed in to the `TPM2_StartAuthSession` command.
    - An unsalted session doesn't add entropy in this way.
- *Session use modifiers:* These modify the actions of an HMAC or policy session on a per-command basis. Continue, encrypt, decrypt, and audit are the more commonly used modifiers:
    - *Continue*: If not set, the session is terminated after a successful command.
    - *Decrypt*: Indicates that the first TPM2B command parameter is sent to the TPM in encrypted form.

- *Encrypt*: Causes the first TPM2B response parameter to be returned from the TPM in encrypted form.
- *Audit*: Causes a command using the session to be audited.

Based on an understanding of these terms, I can now describe the different types of sessions.

Password, HMAC, and Policy Sessions: What Are They?

All three types of sessions are a means of authorizing actions and, in the case of HMAC and policy sessions, configuring sessions on a per-command basis. Password sessions are the simplest type of authorization: a clear text password is passed down to the TPM to authorize an action. This has obvious security issues if the TPM is being accessed remotely; the intended use of password sessions is for local access. In the TPM, there is a single, always-available password session that is used to authorize a single TPM command with no state preserved between subsequent uses. Because of this, the password session never needs to be started.

HMAC authorizations are a way of using a simple password in a more secure manner; once the calling application and the TPM agree on the password (at the time the entity is created or its authorization value is modified), there is never a need to communicate the password again. This one-time communication of the password to the TPM can be accomplished in a secure manner: that is, the password can be communicated to the TPM in encrypted form. An HMAC session accomplishes this greater level of security by using the password (`authValue`, as it's called in the TPM 2.0 specification) as one of the inputs into an HMAC that is calculated on commands and responses. On a command, the calling application calculates the HMAC and inserts it in the command byte stream. When the TPM receives the command byte stream, if the TPM determines that the HMAC is calculated correctly, the action is authorized. On a response, the TPM calculates an HMAC on the response and inserts it into the response byte stream. The caller independently calculates the response HMAC and compares it to the response byte stream's HMAC field. If they match, the response data can be trusted. All this works only if both the calling application and the TPM know and agree on the `authValue`.

HMAC sessions use two nonces—one from the caller (`nonceCaller`) and one from the TPM (`nonceTPM`)—to prevent replay attacks. These nonces factor into the HMAC calculation. Because `nonceTPM` changes for every command that is sent, and the calling application can, if it wants to, change `nonceCaller` on every command, an attacker can't replay command byte streams. Replayed command bytes streams that use HMAC authorization will always fail because the nonces will be different on the replay.

HMAC sessions maintain state during the lifetime of the session and can be used to authorize multiple actions on TPM entities. An HMAC session is started using the `TPM2_StartAuthSession` command. When started, HMAC sessions can be configured as bound vs. unbound and salted vs. unsalted sessions. The combination of these two options results in four variations of HMAC sessions; these four variations determine how the session key and HMACs are calculated.

Policy sessions, otherwise known as *Enhanced Authorization* (EA), are built on top of HMAC sessions and add an extra level of authorization. Whereas HMAC authorizations are based only on an authorization value or password, policy authorizations enhance this with authorizations based on TPM command sequences, TPM state, and external devices such as fingerprint readers, retina scanners, and smart cards, to name a few. Many conditions can be ANDed and ORed together into complex authorization trees, providing unlimited authorization possibilities.

Table 13-1 shows a high-level summary of the various types of authorizations.

Table 13-1. *Comparison of the Three Types of Sessions*

	Password	HMAC	Policy
State/Other Info	No state is maintained between subsequent uses.	State is maintained for the lifetime of the session.	State is maintained for the lifetime of the session. Built on top of HMAC sessions.
Security	The password is in the clear on every command; a snooper could easily grab the password.	Much more secure than a password (especially when sending commands to remote a TPM). Nonces are used to prevent replay attacks.	Enhanced security by allowing complex sequences of commands and internal and external states to authorize. Nonces are used to prevent replay attacks if an HMAC is being used.
Method of Starting	None	`TPM2_StartAuthSession`	`TPM2_StartAuthSession`
Per-Command Session Modifiers	None	Decrypt, encrypt, audit	Decrypt and encrypt

With that under your belt, let's look at some important nuances in how the specification uses the terms *session* and *authorization*. Pay attention here; understanding these will greatly enhance your ability to read and understand the TPM 2.0 specification as well as the rest of this chapter.

Session and Authorization: Compared and Contrasted

Sessions and authorizations are closely related and sometimes overlapping concepts in the TPM 2.0 specification, but they are not synonymous terms. Sessions are the vehicle for authorizations, but they're also used for purposes other than authorization, in conjunction with authorizations or completely independent of them. For example, sessions used for authorization can also be used to specify per-command modifiers such as encrypt, decrypt, and audit. Sessions can also be used for these per-command modifiers without being simultaneously used for any authorizations at all.

The TPM 2.0 specification itself often overlaps the terms *session* and *authorization*. Here are some examples of this in the specification:

- The *authorization area*[1] in commands is used for both authorizations and sessions. But sessions can be used in ways that have nothing to do with authorization. For instance, they can be used to set up encryption and decryption of command and response parameters and to enable `auditing` of commands. Sessions that have nothing to do with authorization can be configured for these purposes.
- The `TPM_ST_NO_SESSIONS` and `TPM_ST_SESSIONS` tags are used to indicate whether an authorization area is present in a command, an obvious lack of consistency in nomenclature.
- Sessions are started with the `TPM2_StartAuthSession` command. The name of the command indicates that an authorization is being started, but in fact a session is being started by this command.[2] The session being started might never be used for authorization.
- Another case is password authorizations. Technically these are sessions, but no state is maintained between subsequent commands, and `TPM2_StartAuthSession` isn't used to start a password "session". A password authorization is a one-shot authorization that applies to only one command.

The reason for noting these aspects is to help you comprehend the specification. Understanding the distinctions between these blurred usages of terms helped me as I was struggling to understand these concepts. As a result, I developed diagrams to help categorize the various types of authorizations, sessions, and session modifiers. Hopefully these will help you, too.

[1] A more technically accurate name for this would have been the *sessions area*.
[2] A more technically accurate name for this command would have been `TPM2_StartSession`.

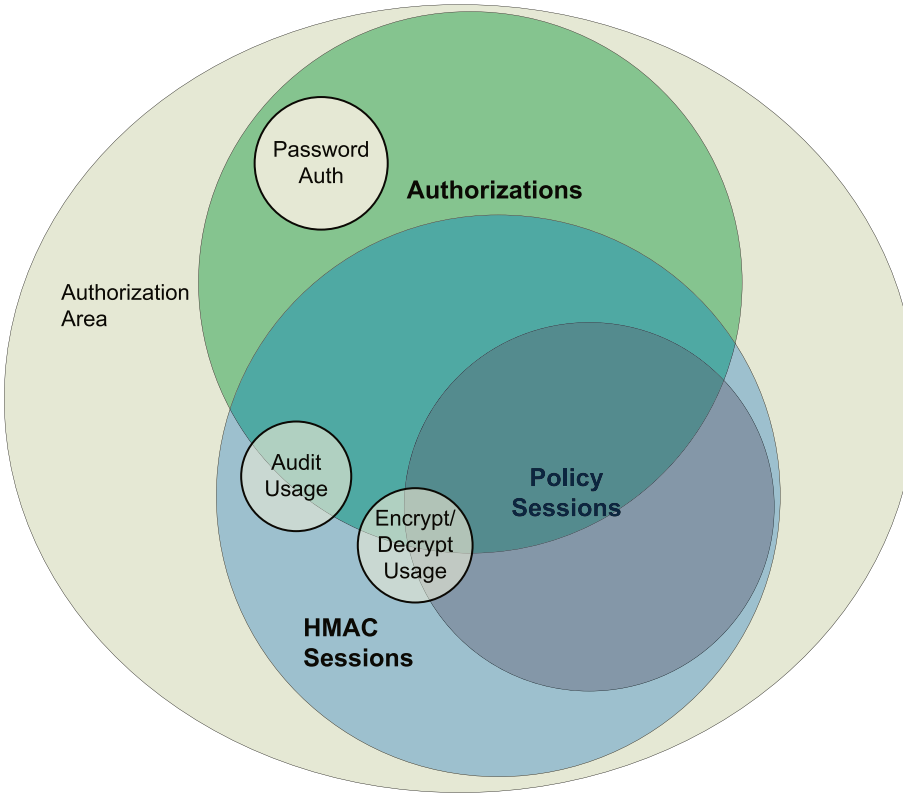

Figure 13-1. *Authorizations and sessions Venn diagram*

The following points are of special note in this diagram:

- Authorizations can be password, HMAC, or policy authorizations.
- Password authorizations can never be used for session use modifiers.

■ **Note** In Figure 13-1, audit, encrypt, or decrypt are the only session use modifiers shown, but there are others. These three are shown because they're the more commonly used ones.

- HMAC and policy sessions can be used for authorizations but can also be used to set session-use modifiers apart from any authorization. This why the HMAC and policy sessions straddle the authorization circle's boundary.
- The command's authorization area is where all of these authorizations, sessions, and session modifiers are specified.
- Command modifiers can be used in sessions used for authorization as well as those that aren't, which is why the audit, encrypt, and decrypt circles straddle the authorization circle's boundary.
- Sessions that aren't used for authorization can also be in the authorization area of the command and response byte streams.
- Policy sessions can be used for encrypt or decrypt, but not for audit.[3]
- HMAC sessions can be used for encrypt, decrypt, and/or audit.

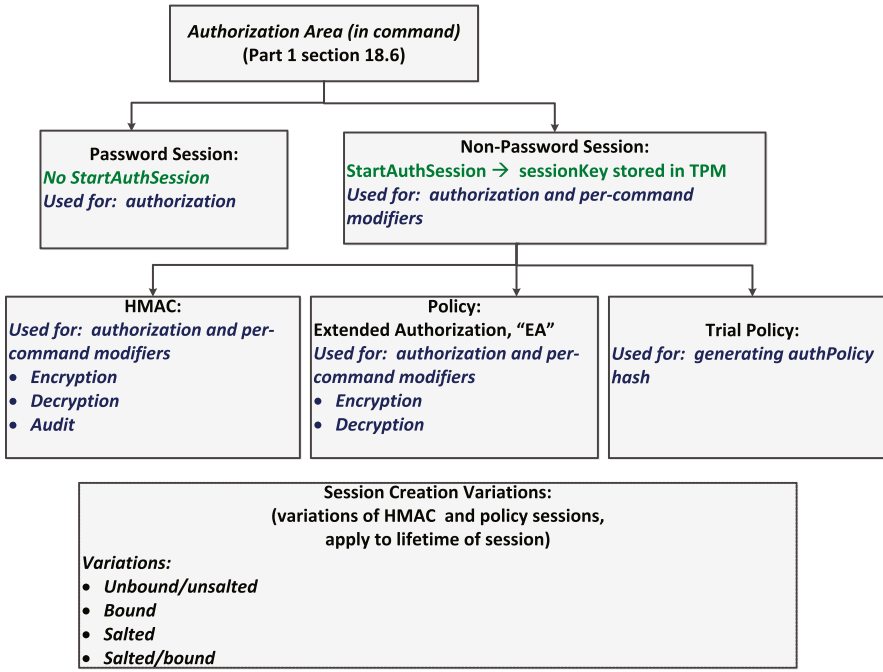

Figure 13-2. *Authorizations and sessions block diagram*

[3]According to the TPM 2.0 specification developers, this was an optimization and not due to any fundamental technical difficulty.

Figure 13-2 illustrates the relationship somewhat differently. Note the following points in this diagram:

- The authorization area can specify the parameters for password, HMAC, or policy sessions. The authorization area is described in detail in the next section.
- Sessions started with the `TPM2_StartAuthSession` command can be HMAC, policy, or trial policy sessions.
- HMAC sessions can be configured on a per-command basis to be audit, decrypt, and/or encrypt sessions.
- Policy sessions can be configured on a per-command basis to be decrypt and/or encrypt sessions. They cannot be used for audit.
- The four session initialization variations can apply to HMAC, policy, or trial policy sessions.

The important things to remember are that sessions are the vehicle for authorizations but can also be used apart from any authorizations for per-command actions that are set by session modifiers.

Regardless of how the sessions are used or the type of authorization, if any, the command and response authorization areas are used to communicate the authorization and session data to and from the TPM. Before I describe the details of command and response areas, you need to understand the functions of authorization roles.

Authorization Roles

Authorization roles for each command are specified in Part 3's descriptions of commands. These roles and the rules related to them act in a manner similar to access control lists (ACLs) for computer directories. Authorization roles control the types of authorizations that can be used to run commands, which essentially means they control who gets to run specific commands and under what circumstances.

There are three possible roles: `USER`, `ADMIN`, and `DUP`. `USER` is used for normal uses of the entity, `ADMIN` role is used for system management tasks, and `DUP`, a narrowly focused role, is the only role allowed for the `TPM2_Duplicate` command.

Two attributes of entities that determine the type of authorization required are `userWithAuth` and `adminWithPolicy`. These attributes either are set explicitly (at object creation time for objects) or determined by other means for certain permanent handles and NV indices:

- `userWithAuth`:
  - `Set` means `USER` role authorization can be provided by a password, HMAC, or policy session.
  - `Clear` means `USER` role authorization must be provided by a policy session.

- `adminWithPolicy`:
  - `Set` means `ADMIN` role authorization must be provided by a policy session.[4]
  - `Clear` means `ADMIN` role authorization can be provided by a password, HMAC, or policy session.

If the authorization role is `ADMIN`:

- For object handles, the required authorization is determined by the object's `adminWithPolicy` attribute, which is set when the object is created.
- For the handles `TPM_RH_OWNER`, `TPM_RH_ENDORSEMENT`, and `TPM_RH_PLATFORM`, the required authorization is as if `adminWithPolicy` is set.
- For NV indices, the required authorization is as if the `adminWithPolicy` attribute was set when the NV index was created.

If authorization role is `USER`:

- For object handles, the required authorization is determined by the object's `userWithAuth` attribute, which is set when the object is created.
- For the handles `TPM_RH_OWNER`, `TPM_RH_ENDORSEMENT`, and `TPM_RH_PLATFORM`, the required authorization is as if `userWithAuth` is set.
- For NV index handles, the required authorization is determined by the following NV index attributes: `TPMA_NV_POLICYWRITE`, `TPMA_NV_POLICYREAD`, `TPMA_NV_AUTHWRITE`, and `TPMA_NV_AUTHREAD`. These attributes are set when the NV index is created.

[4]A more accurate name for this attribute would have been `adminOnlyWithPolicy`.

If the authorization role is DUP:

- The authorization must be a policy authorization.
- The DUP role is only used for objects.

If the authorization role is DUP or ADMIN, the command being authorized must be specified in the policy.

Now that you understand roles, let's look at the authorization area.

Command and Response Authorization Area Details

Chapter 5 described the command and response data schematics but purposely left out one important area in commands and responses: the authorization area. This area is where sessions and authorizations are specified in the command and response byte stream, and a detailed discussion was deferred until this chapter.

To make the concepts more practical, this section examines these two areas using the TPM2_NV_Read command. The same general format is followed for authorization areas for all commands that can have authorization areas.

Command Authorization Area

Figure 13-3 shows the TPM2_NV_Read command and response data schematics and the location of the authorization areas in the command. Note that these areas aren't specifically called out in the Part 3 schematics, but they're implied; this is why they're shown in boxes off to the left side of the command and response schematic tables. For all commands that take authorizations, the authorization area for the command is located after the handles area and before the parameters area. The authorization area for the response is located at the end of the response after the response parameters.

Type	Name	Description
TPMI_ST_COMMAND_TAG	Tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Read
TPMI_RH_NV_AUTH	@authHandle	The handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	The NV Index to be read Auth Index: None
UINT16	size	Number of octets[5] to read
UINT16	offset	Octet offset into the area This value shall be less than or equal to the size of the nvIndex data.

Authorization Area

Type	Name	Description
TPM_ST	Tag	
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_NV_BUFFER	data	The data read

Authorization Area

Figure 13-3. *NV_Read command and response schematic from TPM 2.0 spec, Part 3, and the location of authorization areas. The boxes to the left indicate where the authorization areas are sandwiched in. This is often confusing to new readers of the specification but is very important to grasp.*

For any command that can take authorizations, there can be up to three authorization structures in the authorization area. For a successful TPM 2.0 command, the number of authorization structures in the response is always equal to the number of authorization structures in the command. For a TPM 2.0 command that fails, the number of authorization structures in the response is always 0.

[5]The term *octet* is used in the TPM specification to denote 8 bits, which is often, although somewhat inaccurately, referred to as a *byte*. Because some computers use *bytes* that have a different number of bits, the TPM 2.0 architects used the term *octet*.

For the command, notice the @ sign in front of authHandle: this means an authorization structure is required to authorize actions on the entity corresponding to the authHandle. Further notation in the description column, "Auth role: USER," indicates the authorization role required.

Command Authorization Structures

The command authorization structure, TPMS_AUTH_COMMAND, is illustrated in Figure 13-4. This shows the details of the command authorization area box from Figure 13-3.

session handle	A four-octet value indicating the session number associated with this data block (TPM_RS_PW for a password authorization)
size field	A two-octet value indicating the number of octets in *nonce*
nonce	If present, an octet array that contains a number chosen by the caller
session attributes	A single octet with bit fields that indicate session usage
size field	A two-octet value indicating the number of octets in *authorization*
authorization	If present, an octet array that contains either an HMAC or a password, depending on the session type

Figure 13-4. *Command authorization structure, TPMS_AUTH_COMMAND*

Although not strictly part of the authorization structure in the current TPM 2.0 specification, the authorizationSize field in a command is present if the command tag is TPM_ST_SESSION, which indicates that the authorization area is present. This authorizationSize field allows code that is parsing the command to determine how many sessions are in the authorization area and where to find the parameters. The field immediately precedes the authorization area as shown in Figure 13-5.

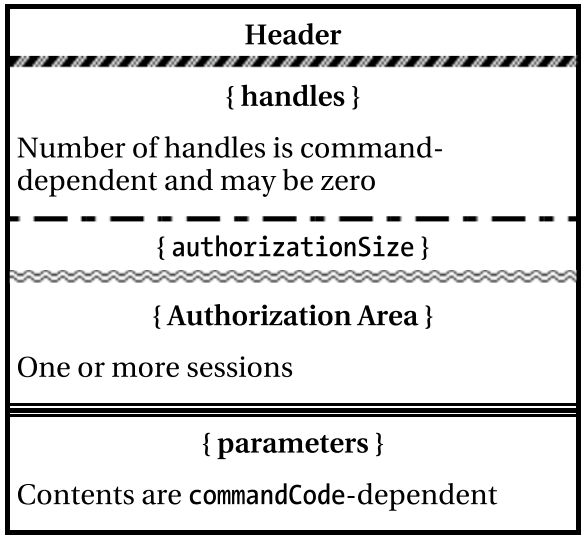

Figure 13-5. *Command structure showing where the command authorization area(s) are located*

Response Authorization Structures

For a response, the authorization structure, `TPMS_AUTH_RESPONSE`, is shown in Figure 13-6. This shows the details of the response authorization area box from Figure 13-3.

size field	A two-octet value indicating the number of octets in *nonce* (zero for a password authorization)
nonce	If present, an octet array that contains a number chosen by the TPM
session attributes	A single octet with bit fields that indicate session usage
size field	A two-octet value indicating the number of octets in *acknowledgment*
acknowledgment	If present, an octet array that contains an HMAC

Figure 13-6. *Response authorization structure, `TPMS_AUTH_RESPONSE`*

The response authorization area is at the very end of the response. To make it easy to find, a `parameterSize` field, a UINT32, is inserted before the response parameter area for all responses that contain an authorization area. Code that is parsing the response can use the `parameterSize` field to skip past the response parameters to find the response authorization area. The `parameterSize` field isn't present when a response doesn't include an authorization area (see Figure 13-7).

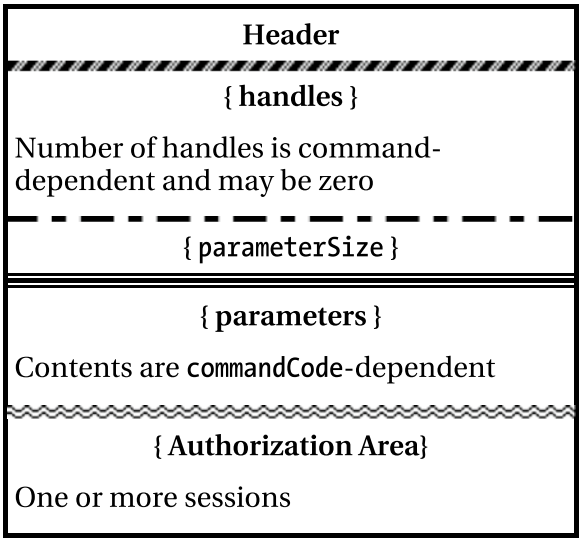

Figure 13-7. *Response Structure, showing where the response authorization area(s) and parameterSize fields are located*

Now that you know what the authorization areas look like, let's look at the three types of authorizations in detail.

Password Authorization: The Simplest Authorization

Password authorizations are the simplest authorizations, so I will describe them first. This section presents the password authorization lifecycle: how to create a password authorized entity, how to alter the authorization for an existing entity, and how to use a password authorization to authorize an action.

Password Authorization Lifecycle

A password authorization has a very simple lifecycle: create an entity using a password as the authorization, and then authorize actions on the entity. In more detail, the high-level steps required to create and use a password authorization are as follows:

1. Create an entity that will use an authorization value, or change the authorization value for an existing entity. This step is typically performed once per entity.
2. Authorize actions using the password authorized entity. This step can be performed multiple times for a particular entity and can occur any time after the entity's password has been set, whether by creating the entity or by changing its authorization.

First let's look at step 1, creating an entity to use a password authorization or altering the password for an existing entity.

Creating a Password Authorized Entity

To create an entity, use the following commands: `TPM2_CreatePrimary`, `TPM2_Create`, and `TPM2_NV_DefineSpace`.[6] Each of these has a parameter field for passing in the `authValue` that will be used to authorize actions on the entity. This `authValue` can be used either as a simple plaintext password or as an input to an HMAC authorization, but since this section is describing a password session, it just describes its use as a password. HMAC authorizations are described after we finish with passwords.

Here are some more details about the three TPM commands used to create entities:

- `TPM2_CreatePrimary` is used to create primary objects (objects directly under the primary seed) in a hierarchy. The `USER` authorization can be a password authorization if the `inPublic` parameter's `userWithAuth` attribute is set; this means authorization for actions that require the `USER` role can be performed by a password or HMAC. The `authValue`, a password in this case, is passed in by setting the `userAuth` field of the `inSensitive` parameter to the password.
- `TPM2_Create` is used to create objects that can be loaded into the TPM. The authorization type, `userWithAuth`, and the `authValue` are configured by setting the same fields used by `TPM2_CreatePrimary`.
- `TPM2_NV_DefineSpace` is used to define an NV index. A password authorization can be used if the attributes `TPMA_NV_AUTHREAD` and/or `TPMA_NV_AUTHWRITE` are set. The input parameter, `authValue` (the password), is passed in as the `auth` parameter of the `TPM2_NV_DefineSpace` command.

Changing a Password Authorization for an Already Created Entity

To change the password of an entity, these commands are used:

- `TPM2_ObjectChangeAuth`: Can be used to change the authorization of objects that aren't primary objects.
- `TPM2_HierarchyChangeAuth`: Used to change the authorization for a hierarchy (platform, owner, or endorsement) or for the lockout authority.
- `TPM2_NV_ChangeAuth`: Changes the authorization value for an NV index.

[6]All of these commands have the ability to set the authorization to use an `authValue` and/or a policy, but only the use of `authValue` is described here.

Now let's look at step 2, authorizing actions using a password authorization.

Using a Password Authorization

A password authorization is the simplest authorization. To use a password authorization, no session needs to be started. The caller simply fills in the command authorization block as shown in Figure 13-8.

Type	Name	Description
TPMI_SH_AUTH_SESSION	authHandle	Required to be the reserved authorization session handle TPM_RS_PW
TPM2B_NONCE	nonce	Required to be an empty buffer
TPMA_SESSION	sessionAttributes	Only continueSession may be set[7]
TPM2B_AUTH	password	Authorization compared to the authValue of the TPM entity

Figure 13-8. *Password authorization command*[7]

The response authorization area looks like Figure 13-9.

Type	Name	Description
TPM2B_NONCE	nonceTPM	Zero-length for a password authorization
TPMA_SESSION	sessionAttributes	Copy of the flags from the password authorization in the command[8]
TPM2B_AUTH	hmac3	Zero-length buffer for a password authorization

Figure 13-9. *Password acknowledgement in response*[8, 9]

Code Example: Password Session

Listing 13-1 shows a code example of a password session using the password session test from the TSS SAPI test code. This code uses the TSS System API that was described in Chapter 7. Because you hadn't yet learned about authorizations and sessions, a description of the authorization-related structures and functions was deferred until this

[7]Because the password session is always available, continueSession has no effect.
[8]The continueSession flag is an exception to this. For password sessions, the continueSession flag is always set in the response.
[9]This field is called hmac in the specification, but it isn't really an HMAC. Actually it's not anything at all, because it's a zero-length buffer. Probably the specification writers called it hmac to keep it consistent with HMAC authorization areas.

chapter. In order to follow the code example, you need to understand these session and authorization related System API data structures:

- `TSS2_SYS_CMD_AUTHS`: Specifies the number of authorization areas for the command and the specific authorization areas to be used. The structure looks like this:

```
typedef struct {
    uint8_t cmdAuthsCount;
    TPMS_AUTH_COMMAND **cmdAuths;
} TSS2_SYS_CMD_AUTHS;
```

- `TSS2_SYS_RSP_AUTHS`: In a like manner, specifies the number of authorization areas in a response and the specific response authorization areas. The structure looks like this:

```
typedef struct {
    uint8_t rspAuthsCount;
    TPMS_AUTH_RESPONSE **rspAuths;
} TSS2_SYS_RSP_AUTHS;
```

■ **Note** The `CheckPassed` and `CheckFailed` functions used in the code example are the same as those described in the code example in the SAPI section of Chapter 7.

Now that you understand the new structures, let's look at the code. I've added notes for each major block of the code to help you follow it better.

Listing 13-1. Password Authorization: Code Example

```
// Password used to authorize access to the NV index.
char password[] = "test password";

// NV Index used for the password test.
#define TPM20_INDEX_PASSWORD_TEST       0x01500020

void PasswordTest()
{
    UINT32 rval;
    int i;
```

Create an authorization area for the command and response.

```
// Authorization structure for command.
TPMS_AUTH_COMMAND sessionData;

// Authorization structure for response.
TPMS_AUTH_RESPONSE sessionDataOut;

// Create and init authorization area for command:
// only 1 authorization area.
TPMS_AUTH_COMMAND *sessionDataArray[1] = { &sessionData };

// Create authorization area for response:
// only 1 authorization area.
TPMS_AUTH_RESPONSE *sessionDataOutArray[1] = { &sessionDataOut };

// Authorization array for command (only has one auth structure).
TSS2_SYS_CMD_AUTHS sessionsData = { 1, &sessionDataArray[0] };

// Authorization array for response (only has one auth structure).
TSS2_SYS_RSP_AUTHS sessionsDataOut = { 1, &sessionDataOutArray[0] };
TPM2B_MAX_NV_BUFFER nvWriteData;

printf( "\nPASSWORD TESTS:\n" );
```

Create an NV index.

```
// Create an NV index that will use password
// authorizations. The password will be
// "test password".
CreatePasswordTestNV( TPM20_INDEX_PASSWORD_TEST, password );

//
// Initialize the command authorization area.
//

// Init sessionHandle, nonce, session
// attributes, and hmac (password).
sessionData.sessionHandle = TPM_RS_PW;

// Set zero sized nonce.
sessionData.nonce.t.size = 0;

// sessionAttributes is a bit field. To initialize
// it to 0, cast to a pointer to UINT8 and
```

```
// write 0 to that pointer.
*( (UINT8 *)&sessionData.sessionAttributes ) = 0;

// Init password (HMAC field in authorization structure).
sessionData.hmac.t.size = strlen( password );
memcpy( &( sessionData.hmac.t.buffer[0] ),
        &( password[0] ), sessionData.hmac.t.size );
```

Do writes, one with the correct password and one with an incorrect one; then verify the results.

```
// Initialize write data.
nvWriteData.t.size = 4;
for( i = 0; i < nvWriteData.t.size; i++ )
    nvWriteData.t.buffer[i] = 0xff - i;

// Attempt write with the correct password.
// It should pass.
rval = Tss2_Sys_NV_Write( sysContext,
        TPM20_INDEX_PASSWORD_TEST,
        TPM20_INDEX_PASSWORD_TEST,
        &sessionsData, &nvWriteData, 0,
        &sessionsDataOut );
// Check that the function passed as
// expected. Otherwise, exit.
CheckPassed( rval );

// Alter the password so it's incorrect.
sessionData.hmac.t.buffer[4] = 0xff;
rval = Tss2_Sys_NV_Write( sysContext,
        TPM20_INDEX_PASSWORD_TEST,
        TPM20_INDEX_PASSWORD_TEST,
        &sessionsData, &nvWriteData, 0,
        &sessionsDataOut );
// Check that the function failed as expected,
// since password was incorrect. If wrong
// response code received, exit.
CheckFailed( rval,
        TPM_RC_S + TPM_RC_1 + TPM_RC_AUTH_FAIL );
```

Delete the NV index.

```
    // Change hmac to null one, since null auth is
    // used to undefine the index.
    sessionData.hmac.t.size = 0;

    // Now undefine the index.
    rval = Tss2_Sys_NV_UndefineSpace( sysContext, TPM_RH_PLATFORM,
            TPM20_INDEX_PASSWORD_TEST, &sessionsData, 0 );
    CheckPassed( rval );
}
```

A good understanding of password authorizations and the data structures used to enable them provides a foundation for understanding the other types of authorizations. Next I describe HMAC and policy authorizations: specifically, how to start them.

Starting HMAC and Policy Sessions

Both HMAC and policy sessions are started using the `TPM2_StartAuthSession` command. When a session is started, it must be one of the following session types: HMAC, policy, or trial policy. Earlier I described HMAC and policy sessions at a high level, but those descriptions didn't mention trial policy sessions. *Trial policy sessions* are neutered policy sessions: they can't authorize any actions, but they can be used to generate policy digests before creating entities (more on that later). For the purposes of this section, policy and trial policy sessions are grouped together.

When a session is started, basic characteristics of the session are determined. Specifically, whether the session is bound or unbound, whether the session is salted or unsalted, the strength of the session key, the strength of the anti-replay protections, the strength of parameter encryption and decryption, and the strength of the session HMACs are determined by the parameters used to call `TPM2_StartAuthSession`.

Some terms need to be understood before this section describes the process of starting HMAC and policy sessions:

- *KDFa*: The key-derivation function used to create session keys.[10] An HMAC function is used as the pseudo-random function for generating the key. The inputs to the KDFa are a hash algorithm; an HMAC key, K (described next); a 4-byte string used to identify the usage of the KDFa output; `contextU` and `contextV` (variable-length strings); and the number of bits in the output. These parameters are cryptographically combined by the KDFa function to create the session key, described below.
- *K*: The key used as input to the KDFa function. For session-key creation, K is the concatenation of the `authValue` (of the entity corresponding to the `bind` handle) and the `salt` parameter passed to the `TPM2_StartAuthSession` command.

[10]KDFa is used for many other things in the TPM, but this section only discusses its use in sessions.

- `sessionKey`: A key created when an HMAC or policy session is started.[11] For session key creation, the KDFa function takes the following as inputs:
  - `sessionAlg` (a hash algorithm)
  - K (the HMAC key used as input to the KDFa's HMAC function)
  - A unique 4 byte label, `ATH` (three characters plus the string terminator)
  - Two nonces, `nonceTPM` and `nonceCaller` (corresponding to `contextU` and `contextV` in the KDFa)
  - The number of bits in the resulting key
- `nonceCaller`: The nonce sent by the caller to the `TPM2_StartAuthSession` command.
- `nonceTpm`: The nonce generated by the TPM in response to the `TPM2_StartAuthSession` command and returned to the caller.

TPM2_StartAuthSession Command

As noted earlier, the parameters to the `TPM2_StartAuthSession` function determine many of the session's characteristics, including the session's security properties. The command schematic for this command is shown in Figure 13-10; the response is shown in Figure 13-11.

[11] It is important to avoid confusing terms here; specifically, `sessionKey` should not be confused with `hmacKey`. The `hmacKey` isn't determined at session creation time, but it's partially determined by the parameters used to start the session.

Type	Name	Description
TPMI_ST_COMMAND_TAG	Tag	TPM_ST_SESSIONS if an audit, decrypt, or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	Handle of a loaded decrypt key used to encrypt salt May be TPM_RH_NULL Auth Index: None
TPMI_DH_ENTITY+	Bind	Entity providing the authValue May be TPM_RH_NULL Auth Index: None
TPM2B_NONCE	nonceCaller	Initial nonceCaller;sets the nonce size for the session Must be at least 16 octets
TPM2B_ENCRYPTED_SECRET	encryptedSalt	Value encrypted according to the type of tpmKey If tpmKey is TPM_RH_NULL, this must be the empty buffer.
TPM_SE	sessionType	Indicates the type of the session: simple HMAC or policy (including a trial policy)
TPMT_SYM_DEF+	Symmetric	Algorithm and key size for parameter encryption May select TPM_ALG_NULL
TPMI_ALG_HASH	authHash	Hash algorithm to use for the session Must be a hash algorithm supported by the TPM and not TPM_ALG_NULL

Figure 13-10. *TPM2_StartAuthSession command*

Type	Name	Description
TPM_ST	Tag	
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_SH_AUTH_SESSION	sessionHandle	Handle for the newly created session
TPM2B_NONCE	nonceTPM	The initial nonce from the TPM, used in the computation of sessionKey

Figure 13-11. *TPM2_StartAuthSession response*

This command takes the following handles and parameters as inputs:

- Two handles:
  - If tpmKey is TPM_RH_NULL, the session is an unsalted session; otherwise, it's a salted session, and the encryptedSalt parameter is decrypted by the TPM to get the salt value used to add entropy. The TPM uses the loaded key pointed to by the tpmKey handle to do the decryption of encryptedSalt.
  - If bind is TPM_RH_NULL, the session is an unbound session. Otherwise, it's a bound session, and the authValue of the entity pointed to by the bind handle is concatenated with the salt value to form K, which is used in calculating the sessionKey.

- Five parameters:
  - `nonceCaller` is the first nonce set by the caller and sets the size for all subsequent nonces returned by the TPM.
  - `encryptedSalt` is used only if the session is salted as described earlier in the discussion of `tpmKey`. If the session is unsalted, this parameter must be a zero-sized buffer.
  - `sessionType` determines the type of the session: HMAC, policy, or trial policy.
  - `symmetric` determines the type of parameter encryption that will be used when the session is set for encrypt or decrypt.
  - `authHash` is the algorithm ID for the hash algorithm that will be used by the session for HMAC operations.

When a session is started, the TPM processes the command and generates a session handle, computes a `nonceTPM`, and calculates a session key. This key is used to generate HMACs, encrypt command parameters, and decrypt response parameters. After the session is created, the session key remains the same for the lifetime of the session. The session handle and the `nonceTPM` are returned by the command.

The session key is determined by these command parameters passed in to `TPM2_StartAuthSession`: `tpmKey`, `bind`, `encryptedSalt`, `nonceCaller`, and `authHash`. The response parameter, `nonceTPM`, also figures into the session key.[12] Use of the `nonceTPM` in creating the session key guarantees that using the same `authValue`, `salt`, and `nonceCaller` will generate a different session key.

Because the calling application also has to know the session key, it duplicates the TPM's calculations using the `nonceTPM` along with the input variables to perform this calculation. At this point, the session has started, and both the caller and the TPM know the session key.

Session Key and HMAC Key Details

Table 13-2 describes the variations of sessions and how the `sessionKey` and HMAC key are created for each case. Having all this information in a single table can be very helpful, which is why it's included here.

[12] The symmetric parameter to `TPM2_StartAuthSession` is only used for encryption and decryption of command and response parameters, so it isn't described in this chapter.

Table 13-2. *Variations of Sessions and Session Key Creation*

Session Variation	bind	tpmKey	K	sessionKey	HMAC key
Unbound/ Unsalted	TPM_RH_NULL	TPM_RH_NULL	Null	NULL key	Entity authorization value, $authValue_{entity}$
Bound	Not TPM_RH_NULL	TPM_RH_NULL	bind entity's authorization value, $authValue_{bind}$	KDFa (sessionAlg, $authValue_{bind}$, "ATH", nonceTPM, nonceCaller, bits)[13]	if entity == bind entity AND not a policy session: sessionKey if entity != bind entity OR session is a policy session[14]: sessionKey || $authValue_{entity}$
Salted	TPM_RH_NULL	Not TPM_RH_NULL	salt	KDFa(sessionAlg, salt, "ATH", nonceTPM, nonceCaller, bits)	sessionKey || $authValue_{entity}$
Bound/Salted	Not TPM_RH_NULL	Not TPM_RH_NULL	$authValue_{bind}$ || salt	KDFa (sessionAlg, ($authValue_{bind}$ || salt), "ATH", nonceTPM, nonceCaller, bits)	If(entity == bind entity AND not a policy session: sessionKey if entity != bind entity OR session is a policy session: sessionKey || $authValue_{entity}$

[13]Including the two nonces, nonceCaller and nonceTPM, in the session-key creation makes it statistically impossible to create two sessions with the same session key. This property of TPMs enables security analysis.

[14]A policy session always acts as if it's an unbound session.

Guidelines for TPM2_StartAuthSession Handles and Parameters

From the details in Table 13-2, we can deduce some guidelines for choosing the `TPM2_StartAuthSession` parameters. The strength of the session key is determined by the combination of the `bind` and `tpmKey` handles, `encryptedSalt`, `nonceCaller`, and the hash algorithm used for the session.

The strongest possible session key is provided with the `bind` handle pointing to a TPM entity (bound session), the `tpmKey` handle pointing to a loaded key (salted session), and `nonceCaller`'s size set to the size of the hash algorithm's output.

With `bind` and `tpmKey` set to `TPM_RH_NULL`, the result is a zero-length session key—a very weak session key. However, as long as the entity's `authValue` is strong, the HMAC key is still strong. As will be detailed in Chapter 17, the strength of the session key directly affects the strength of the encryption and decryption of the command and response parameters.

The length of the `nonceCaller` parameter determines the length of the `nonceTPM`s used in the session. The bigger the nonce, the better the protection against replay attacks.

The session key and the entity's authorization value are used in generating session HMACs, so again, a stronger session key and stronger authorization value result in greater security.

Programmers making calls to `TPM2_StartAuthSession` should consider carefully which properties are desired when selecting the parameters to use.

Session Variations

Now let's examine the meaning of bound vs. unbound sessions and salted vs. unsalted sessions in detail. I will also describe some use cases for them.

Salted vs. Unsalted

Both HMAC and policy sessions can be salted or unsalted. A salted session adds more entropy to the session key creation. Whether a session is salted or not is determined by the `tpmKey` parameter to the `TPM2_StartAuthSession` command. This decrypted salt is added into the session key creation process. If the `authValue` is weak, salting the session helps to prevent offline hammering attacks. An *offline hammering attack* consists of trying different values of `authValue` to see if the correct HMAC can be generated. If successful, the `authValue` has been discovered. Salting of sessions raises the bar for this type of attack.

Bound vs. Unbound

Similarly, both HMAC and policy sessions can be set to be either bound or unbound. A *bound* session means the session is "bound" to a particular entity, the "bind" entity; a session started this way is typically used to authorize multiple actions on the bind entity. The bind entity's authorization value is used to calculate the session key but isn't needed after that. This can be advantageous from a security perspective, because the calling

application doesn't need to keep prompting for the authorization value (password) or maintain it in memory.

Bound sessions can also be used to authorize actions on other entities, and in that case, the bind entity's `authValue` adds entropy to the session key creation, resulting in stronger encryption of command and response parameters—sort of a poor man's salt. The authorization values for both the bind entity and the entity being authorized figure into the HMAC calculation.

An *unbound* session is used to authorize actions on many different entities. A policy session is most commonly configured as an unbound session. With the security offered by policy sessions, an HMAC isn't as important, and using policy sessions without having to calculate and insert HMACs is much easier.

Use Cases for Session Variations

Now let's answer the obvious question: what are the major use cases for bound/unbound and salted/unsalted sessions? There are many possibilities, but the most common ones are as follows:

- *Unbound sessions* are most commonly used for two cases:
  - If the session is also unsalted, this combination is often used for policy sessions that don't require an HMAC. This is okay because policy sessions use policy commands and HMAC authorization isn't really required in many cases. This simplifies the use of the policy session by eliminating the overhead of calculating the HMACs. The use case for this is any policy authorization that doesn't include the `TPM2_PolicyAuthValue` command.
  - They can also be used by HMAC sessions to authorize actions on many different entities.
- *Bound sessions* have two cases:
  - *Authorizing actions on the bind entity*: This HMAC authorization can be used to authorize many actions on the `bind` entity without prompting for the password each time. For example, an employee might want to view their personnel file many times; this type of authorization would work for that.
  - *Authorizing actions on an entity other than the bind entity*: In this case, both the `bind` entity's `authValue` and the `authValue` of the entity being authorized figure into the HMAC calculation. This results in a stronger session key and stronger encryption and decryption keys.

- *Unsalted session*: when the `authValue` of the `bind` entity is deemed strong enough to generate strong session and strong encryption and decryption keys. If a system administrator can enforce sufficient controls on the strength of a password, an unsalted session using that password may be sufficient.
- *Salted session*: when the `authValue` isn't considered strong enough for generating secure session and encryption/decryption keys. A web site could request two different passwords from a user: one to be used as the authorization value for use of an encryption key, and the other to be used for the salt. The combination of the two would be much stronger than using a single password, as long asa cryptographically strong salt was used.

Now that you have a foundation for starting sessions, let's see some differences between HMAC and policy sessions.

HMAC and Policy Sessions: Differences

HMAC and policy sessions differ primarily in how actions are authorized. Commands sent using HMAC sessions are successful only if the HMAC sent with the command is correct. In order to generate the correct HMAC, knowledge of a secret (`authValue`) that is shared between the caller and TPM is required. In other words, knowledge of the session key and `authValue` enable the calculation of the correct HMAC, effectively granting authorization to perform an action on the entity. An agent that doesn't know either the session key or the `authValue` can't calculate the correct HMAC, which causes the command to fail.

Policy sessions authorize actions based on the correct sequence of policy commands and, in many cases, conditions required by those commands. This is a very simple description of this rich and complicated type of authorization. The details are described in Chapter 14, but suffice it to say that policy sessions authorize actions using the following:

- A sequence of policy commands before the command whose action is being authorized. The presence of this sequence is proven by checking the `policyDigest`. Each policy command hash-extends policy command-specific data into the session's `policyDigest`. In the simplest case, a comparison between the session's `policyDigest` and that of the entity being accessed will determine whether the proper policy commands were performed beforehand.
- A set of conditions that must be met before and/or during the execution of the command whose action is being authorized. If these conditions aren't met, the policy commands fail or the command being authorized fails. This is described in detail later.

Interestingly enough, policy sessions can still have an HMAC in their authorization areas, even though the most common use of policy sessions doesn't, according to Part 1 of the TPM 2.0 specification. This most common use assumes that the session is unbound and unsalted. But when an HMAC is used in the authorization area (whether because the session is bound and/or salted or the `TPM2_PolicyAuthValue` command is used), contrary to HMAC sessions, the HMAC is always calculated as if the session isn't bound to any entity.[15] In policy sessions, the `bind` entity's `authValue` is only used for session key creation and never for HMAC calculation. Applications using the TPM need to account for this during HMAC calculation.

To summarize, HMAC authorizations are more secure than password authorizations, and policy authorizations are the most complex and rich authorizations. HMAC authorizations use a properly calculated HMAC as the means to prove knowledge of the authorization secret(s). Policy authorizations require a set of policy commands and a specific set of conditions required by those policy commands in order to authorize an action. In both HMAC and policy authorizations, HMACs can be used to guarantee command and response integrity.

Now let's look at HMAC authorization in detail.

HMAC Authorization

This section dives into the details of HMAC authorizations. It describes the high-level HMAC authorization lifetime and each of the steps in that lifetime: entity creation or alteration, HMAC session creation, and HMAC session use. The section ends with a description of the security properties of an HMAC session.

As you read this section, I recommend that you reference the example code section. The discussion refers to line numbers in the code where applicable. This section mainly focuses on describing the steps leading up to and including the NV index's write. The NV index's read code is very similar, and mapping of these steps to that code is left as a reader exercise.

HMAC Authorization Lifecycle

The steps for creating and authorizing actions on HMAC authorized entities are the following:

1. Create the entity that will use an authorization value, or change the authorization value for an existing entity. This step is typically performed once per entity.
2. Create an HMAC session.
3. Use the HMAC session to perform operations on the entity. This operation can occur any time after steps 1 and 2 and can occur multiple times. A single HMAC session can be used to authorize multiple actions.

[15]This is an optimization: the session context space normally used for the `bind` value is used for policy-specific parameters.

Altering or Creating an Entity That Requires HMAC Authorization

For the purposes of entity creation, the method of specifying the `authValue` is exactly the same as described earlier in the password authorization lifecycle. The same is true for altering the `authValue` for an existing entity. In both of these operations, the `authValue` is treated exactly the same for HMAC and password authorizations.

In the example code, lines 19–26, 42–44, and 55 set up the `authValue` and `authPolicy` for creating the NV index. Lines 101, 104–105, and 112 set up the NV attributes. And lines 115–117 create the NV index that we're going to authorize.

Creating an HMAC Session

An HMAC session is started with a `TPM2_StartAuthSession` command that has the `sessionType` field set to `TPM_SE_HMAC`. When the HMAC session is started, the TPM creates a session key using the formula described previously. This session key is created in the TPM. After `TPM2StartAuthSession` returns, the caller also recreates the session key, using the `bind` entity's `authValue`, the `salt`, and the `nonceCaller` parameters sent to the TPM by the `TPM2StartAuthSession` command, and the `nonceTPM` returned by the TPM.

Lines 140, 143, and 150 set up the parameters for starting the session, and lines 154–156 actually create the session.

Using an HMAC Session to Authorize a Single Command

The mechanics of a single command during an HMAC session are described in Figure 13-12.

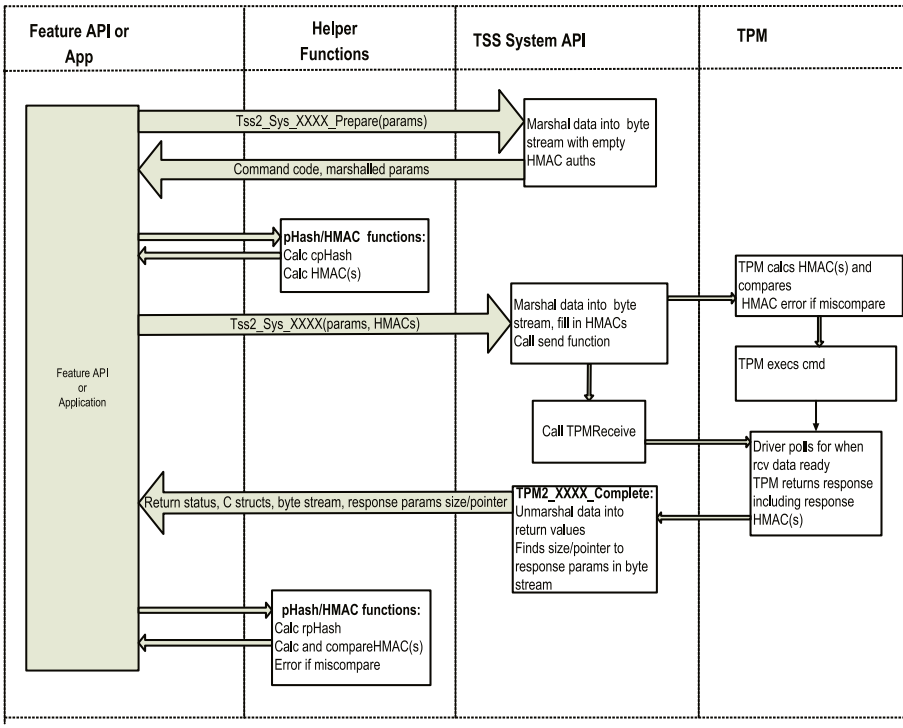

Figure 13-12. *HMAC session: single command. Note that this diagram assumes the use of the TSS SAPI layer. The TAB and resource manager layers are omitted for simplicity. Also, this diagram shows how HMAC sessions operate using the TSS SAPI Tss2_Sys_XXXX_Prepare and one-call interfaces*

To use an HMAC session for authorizing commands, the steps are as follows (see Figure 13-12 while reading the example code):

1. The input parameters are marshalled and concatenated into a single sized byte buffer, `cpParams`. The `Tss2_Sys_NV_Write_Prepare` call on lines 183–185 performs this task and puts the `cpParams` buffer into the `sysContext` structure.

2. The caller calculates the `cpHash`, a hash of the marshalled command parameters contained in the `cpParams` buffer. This is done in the `ComputeCommandHmacs` call on lines 202–205.

3. The caller calculates an HMAC for the command. The `cpHash` is one of the inputs to this calculation. This is done by the `ComputeCommandHmacs` call on lines 202–205.

4. The calculated HMAC is copied into the HMAC session's HMAC field. This is done automatically by the `ComputeCommandHmacs` call—-notice the pointer to `nvCmdAuths` being passed in on lines 202–205.

5. The complete command including header, sessions, and parameters must be marshalled into a byte stream and sent to the TPM. This is done in the one-call function call at lines 211–214.

6. The response must be read from the TPM. This is also done in the one-call function on lines 211–214.

7. After receiving the response, the caller calculates the `rpHash`, a hash of the marshalled response parameters in the byte stream. This is done in the `CheckResponseHmacs` call on lines 224–226.

8. The caller calculates the expected response HMAC. The `rpHash` is one of the inputs to this calculation. This is also done by the `CheckResponseHmacs` call on lines 224–226.

9. The caller compares this calculated response HMAC to the HMAC field of the response's HMAC session. If they aren't the same, the response parameters have been corrupted and none of the data can be trusted. If they are the same, then the response parameters have been received correctly. This is performed by `CheckResponseHmacs`, lines 224–226. It calculates what the response HMAC should be and compares it to the HMAC returned in `nvRspAuths`.

10. If the response HMAC is correct, the response parameters can be unmarshalled into C structures for use by the caller; this is performed by the one-call function on lines 211–214. Note that for the one-call, the code assumes that the HMAC is correct and unmarshals the response parameters. Later, if the response HMAC is proven incorrect, the unmarshalled response parameters can be ignored.

HMAC and Policy Session Code Example

Listing 13-2 presents a simple example of how to execute HMAC and policy sessions. This function is known to work, and all of its support routines are available in the TSS SAPI test code described in Chapter 7. To keep the code as simple as possible, it uses an unbound and unsalted session. If you'd like to see more complicated examples, all variations of bound/unbound and salted/unsalted are tested in the `HmacSessionTest` in the TSS SAPI test code.

■ **Note** Managing HMAC sessions and calculating HMAC authorizations are complicated tasks. Some of the functions called in Listing 13-2 are only explained at a high level. The goal was to demonstrate the high-level flow of an HMAC authorization without overwhelming you with low-level details. If you want to dig deeper, the source code for all the subroutines is available at the web site noted for the TSS SAPI test code in Chapter 7.

To help compare HMAC and policy sessions, this code does HMAC or policy authorizations, depending on the value of the `hmacTest` input parameter; `if` conditional statements using the `hmacTest` parameter are used for all the HMAC- or policy-specific code. For now, because this section is about HMAC authorizations, ignore all the parts of the code that only pertain to policy sessions (these areas are shaded).

Some notes about this code:

- The code does a write to an NV index followed by a read of the same NV index. Both the read and write are authorized using an HMAC authorization.
- To authorize the read and write operations, this code uses either an HMAC session or a policy session with a `TPM2_PolicyAuthValue` command. This provides similar capability (both sessions use an HMAC for authorization), and thus provides a useful vehicle for comparing HMAC and policy sessions.
- The `RollNonces` function does what it says: it copies `nonceNewer` to `nonceOlder` and copies the new nonce to `nonceNewer`. The nonces must be rolled before each command and after each response. This is described more in the section "Using an HMAC Session to Send Multiple Commands (Rolling Nonces)." Here's the complete code for this function:

```
void RollNonces( SESSION *session, TPM2B_NONCE *newNonce  )
{
    session->nonceOlder = session->nonceNewer;
    session->nonceNewer = *newNonce;
}
```

This code example uses a single byte `nonceCaller` for both the NV write and read commands. This isn't a recommended usage—typically, to maximize replay protection, you would use a nonce that is the size of the session's selected hash algorithm, and you would use different randomly generated nonces for each command being authorized.

- This code relies heavily on an application-level structure, SESSION, that maintains all session state information including nonces. There are many ways this can be done—this just happens to be the implementation I chose. This structure looks like this:

```
typedef struct {
    // Inputs to StartAuthSession; these need to be saved
    // so that HMACs can be calculated.
    TPMI_DH_OBJECT tpmKey;
    TPMI_DH_ENTITY bind;
    TPM2B_ENCRYPTED_SECRET encryptedSalt;
    TPM2B_MAX_BUFFER salt;
    TPM_SE sessionType;
    TPMT_SYM_DEF symmetric;
    TPMI_ALG_HASH authHash;

    // Outputs from StartAuthSession; these also need
    // to be saved for calculating HMACs and
    // other session related functions.
    TPMI_SH_AUTH_SESSION sessionHandle;
    TPM2B_NONCE nonceTPM;

    // Internal state for the session
    TPM2B_DIGEST sessionKey;
    TPM2B_DIGEST authValueBind; // authValue of bind object
    TPM2B_NONCE nonceNewer;
    TPM2B_NONCE nonceOlder;
    TPM2B_NONCE nonceTpmDecrypt;
    TPM2B_NONCE nonceTpmEncrypt;
    TPM2B_NAME name;    // Name of the object the session handle
                        // points to. Used for computing HMAC for
                        // any HMAC sessions present.
                        //
    void *hmacPtr;      // Pointer to HMAC field in the marshalled
                        // data stream for the session.
                        // This allows the function to calculate
                        // and fill in the HMAC after marshalling
                        // of all the inputs is done.
                        //
                        // This is only used if the session is an
                        // HMAC session.
                        //
    UINT8 nvNameChanged;// Used for some special case code
                        // dealing with the NV written state.
} SESSION;
```

- `StartAuthSessionWithParams` starts the session, saves its state in a `SESSION` structure, and adds the `SESSION` structure to the application's list of open sessions.
- `EndAuthSession` is used to remove the `SESSION` structure from the application's list of open sessions after the session has ended.

Listing 13-2. Simple HMAC and Policy Code Example

```
void SimpleHmacOrPolicyTest( bool hmacTest )
{
    UINT32 rval, sessionCmdRval;
    TPM2B_AUTH  nvAuth;
    SESSION nvSession, trialPolicySession;
    TPMA_NV nvAttributes;
    TPM2B_DIGEST authPolicy;
    TPM2B_NAME nvName;
    TPM2B_MAX_NV_BUFFER nvWriteData, nvReadData;
    UINT8 dataToWrite[] = { 0x00, 0xff, 0x55, 0xaa };
    char sharedSecret[] = "shared secret";
    int i;
    TPM2B_ENCRYPTED_SECRET encryptedSalt;
    TPMT_SYM_DEF symmetric;
    TPMA_SESSION sessionAttributes;
    TPM_SE tpmSe;
    char *testString;
```

Set up authorizations for NV index creation and deletion.

```
    // Command authorization area: one password session.
    TPMS_AUTH_COMMAND nvCmdAuth = { TPM_RS_PW, };
    TPMS_AUTH_COMMAND *nvCmdAuthArray[1] = { &nvCmdAuth };
    TSS2_SYS_CMD_AUTHS nvCmdAuths = { 1, &nvCmdAuthArray[0] };

    // Response authorization area.
    TPMS_AUTH_RESPONSE nvRspAuth;
    TPMS_AUTH_RESPONSE *nvRspAuthArray[1] = { &nvRspAuth };
    TSS2_SYS_RSP_AUTHS nvRspAuths = { 1, &nvRspAuthArray[0] };

    if( hmacTest )
            testString = "HMAC";
    else
            testString = "POLICY";

    printf( "\nSIMPLE %s SESSION TEST:\n", testString );

```

```
35      // Create sysContext structure.
36      sysContext = InitSysContext( 1000, resMgrTctiContext, &abiVersion );
37      if( sysContext == 0 )
38      {
39          InitSysContextFailure();
40      }
```

Create the NV index, with either an HMAC or a policy authorization required.

```
41      // Setup the NV index's authorization value.
42      nvAuth.t.size = strlen( sharedSecret );
43      for( i = 0; i < nvAuth.t.size; i++ )
44          nvAuth.t.buffer[i] = sharedSecret[i];
45
46      //
47      // Create NV index.
48      //
49      if( hmacTest )
50      {
51          // Set NV index's authorization policy
52          // to zero sized policy since we won't be
53          // using policy to authorize.
54
55          authPolicy.t.size = 0;
56      }
57      else
58      {
59
60          // Zero sized encrypted salt, since the session
61          // is unsalted.
62
63          encryptedSalt.t.size = 0;
64
65          // No symmetric algorithm.
66          symmetric.algorithm = TPM_ALG_NULL;
67
68          //
69          // Create the NV index's authorization policy
70          // using a trial policy session.
71          //
72          rval = StartAuthSessionWithParams( &trialPolicySession,
73                  TPM_RH_NULL, TPM_RH_NULL, &encryptedSalt,
74                  TPM_SE_TRIAL,
75                  &symmetric, TPM_ALG_SHA256 );
76          CheckPassed( rval );
77
```

```
78          rval = Tss2_Sys_PolicyAuthValue( sysContext,
79                  trialPolicySession.sessionHandle, 0, 0 );
80          CheckPassed( rval );
81
82          // Get policy digest.
83          rval = Tss2_Sys_PolicyGetDigest( sysContext,
84                  trialPolicySession.sessionHandle,
85                  0, &authPolicy, 0 );
86          CheckPassed( rval );
87
88          // End the trial session by flushing it.
89          rval = Tss2_Sys_FlushContext( sysContext,
90                  trialPolicySession.sessionHandle );
91          CheckPassed( rval );
92
93          // And remove the trial policy session from
94          // sessions table.
95          rval = EndAuthSession( &trialPolicySession );
96          CheckPassed( rval );
97      }
98
99      // Now set the NV index's attributes:
100     // policyRead, authWrite, and platormCreate.
101     *(UINT32 *)( &nvAttributes ) = 0;
102     if( hmacTest )
103     {
104         nvAttributes.TPMA_NV_AUTHREAD = 1;
105         nvAttributes.TPMA_NV_AUTHWRITE = 1;
106     }
107     else
108     {
109         nvAttributes.TPMA_NV_POLICYREAD = 1;
110         nvAttributes.TPMA_NV_POLICYWRITE = 1;
111     }
112     nvAttributes.TPMA_NV_PLATFORMCREATE = 1;
113
114     // Create the NV index.
115     rval = DefineNvIndex( TPM_RH_PLATFORM, TPM_RS_PW,
116             &nvAuth, &authPolicy, TPM20_INDEX_PASSWORD_TEST,
117             TPM_ALG_SHA256, nvAttributes, 32  );
118     CheckPassed( rval );
119
120     // Add index and associated authorization value to
121     // entity table. This helps when we need
122     // to calculate HMACs.
123     AddEntity( TPM20_INDEX_PASSWORD_TEST, &nvAuth );
124     CheckPassed( rval );
```

```
125
126       // Get the name of the NV index.
127       rval = (*HandleToNameFunctionPtr)(
128               TPM20_INDEX_PASSWORD_TEST,
129               &nvName );
130       CheckPassed( rval );
```

Start the HMAC or policy session.

```
131       //
132       // Start HMAC or real (non-trial) policy authorization session:
133       // it's an unbound and unsalted session, no symmetric
134       // encryption algorithm, and SHA256 is the session's
135       // hash algorithm.
136       //
137
138       // Zero sized encrypted salt, since the session
139       // is unsalted.
140       encryptedSalt.t.size = 0;
141
142       // No symmetric algorithm.
143       symmetric.algorithm = TPM_ALG_NULL;
144
145       // Create the session, hmac or policy depending
146       // on hmacTest.
147       // Session state (session handle, nonces, etc.) gets
148       // saved into nvSession structure for later use.
149       if( hmacTest )
150           tpmSe = TPM_SE_HMAC;
151       else
152           tpmSe = TPM_SE_POLICY;
153
154       rval = StartAuthSessionWithParams( &nvSession, TPM_RH_NULL,
155               TPM_RH_NULL, &encryptedSalt, tpmSe,
156               &symmetric, TPM_ALG_SHA256 );
157       CheckPassed( rval );
158
159       // Get the name of the session and save it in
160       // the nvSession structure.
161       rval = (*HandleToNameFunctionPtr)( nvSession.sessionHandle,
162               &nvSession.name );
163       CheckPassed( rval );
```

Do an NV write using either an HMAC or a policy authorization.

```
164       // Initialize NV write data.
165       nvWriteData.t.size = sizeof( dataToWrite );
166       for( i = 0; i < nvWriteData.t.size; i++ )
167       {
168           nvWriteData.t.buffer[i] = dataToWrite[i];
169       }
170
171       //
172       // Now setup for writing the NV index.
173       //
174       if( !hmacTest )
175       {
176           // Send policy command.
177           rval = Tss2_Sys_PolicyAuthValue( sysContext,
178                   nvSession.sessionHandle, 0, 0 );
179           CheckPassed( rval );
180       }
181
182       // First call prepare in order to create cpBuffer.
183       rval = Tss2_Sys_NV_Write_Prepare( sysContext,
184               TPM20_INDEX_PASSWORD_TEST,
185               TPM20_INDEX_PASSWORD_TEST, &nvWriteData, 0 );
186       CheckPassed( rval );
187
188       // Configure command authorization area, except for HMAC.
189       nvCmdAuths.cmdAuths[0]->sessionHandle =
190               nvSession.sessionHandle;
191       nvCmdAuths.cmdAuths[0]->nonce.t.size = 1;
192       nvCmdAuths.cmdAuths[0]->nonce.t.buffer[0] = 0xa5;
193       *( (UINT8 *)(&sessionAttributes ) ) = 0;
194       nvCmdAuths.cmdAuths[0]->sessionAttributes = sessionAttributes;
195       nvCmdAuths.cmdAuths[0]->sessionAttributes.continueSession = 1;
196
197       // Roll nonces for command
198       RollNonces( &nvSession, &nvCmdAuths.cmdAuths[0]->nonce );
199
200       // Complete command authorization area, by computing
201       // HMAC and setting it in nvCmdAuths.
202       rval = ComputeCommandHmacs( sysContext,
203               TPM20_INDEX_PASSWORD_TEST,
204               TPM20_INDEX_PASSWORD_TEST, &nvCmdAuths,
205               TPM_RC_FAILURE );
206       CheckPassed( rval );
207
208       // Finally!!  Write the data to the NV index.
209       // If the command is successful, the command
210       // HMAC was correct.
```

```
211       sessionCmdRval = Tss2_Sys_NV_Write( sysContext,
212               TPM20_INDEX_PASSWORD_TEST,
213               TPM20_INDEX_PASSWORD_TEST,
214               &nvCmdAuths, &nvWriteData, 0, &nvRspAuths );
215       CheckPassed( sessionCmdRval );
```

Get the response from the NV write. If it's an HMAC session, verify the response HMAC.

```
216       // Roll nonces for response
217       RollNonces( &nvSession, &nvRspAuths.rspAuths[0]->nonce );
218
219       if( sessionCmdRval == TPM_RC_SUCCESS )
220       {
221           // If the command was successful, check the
222           // response HMAC to make sure that the
223           // response was received correctly.
224           rval = CheckResponseHMACs( sysContext, sessionCmdRval,
225                   &nvCmdAuths, TPM20_INDEX_PASSWORD_TEST,
226                   TPM20_INDEX_PASSWORD_TEST, &nvRspAuths );
227           CheckPassed( rval );
228       }
229
230       if( !hmacTest )
231       {
232           // Send policy command.
233           rval = Tss2_Sys_PolicyAuthValue( sysContext,
234                   nvSession.sessionHandle, 0, 0 );
235           CheckPassed( rval );
236       }
```

Do an NV read, using an HMAC or a policy session. If it's an HMAC session, verify the response HMAC. Finally, test the read data against the write data to make sure they're equal.

```
237       // First call prepare in order to create cpBuffer.
238       rval = Tss2_Sys_NV_Read_Prepare( sysContext,
239               TPM20_INDEX_PASSWORD_TEST,
240               TPM20_INDEX_PASSWORD_TEST,
241               sizeof( dataToWrite ), 0 );
242       CheckPassed( rval );
243
244       // Roll nonces for command
245       RollNonces( &nvSession, &nvCmdAuths.cmdAuths[0]->nonce );
246
```

```
247       // End the session after next command.
248       nvCmdAuths.cmdAuths[0]->sessionAttributes.continueSession = 0;
249
250       // Complete command authorization area, by computing
251       // HMAC and setting it in nvCmdAuths.
252       rval = ComputeCommandHmacs( sysContext,
253               TPM20_INDEX_PASSWORD_TEST,
254               TPM20_INDEX_PASSWORD_TEST, &nvCmdAuths,
255               TPM_RC_FAILURE );
256       CheckPassed( rval );
257
258       // And now read the data back.
259       // If the command is successful, the command
260       // HMAC was correct.
261       sessionCmdRval = Tss2_Sys_NV_Read( sysContext,
262               TPM20_INDEX_PASSWORD_TEST,
263               TPM20_INDEX_PASSWORD_TEST,
264               &nvCmdAuths, sizeof( dataToWrite ), 0,
265               &nvReadData, &nvRspAuths );
266       CheckPassed( sessionCmdRval );
267
268       // Roll nonces for response
269       RollNonces( &nvSession, &nvRspAuths.rspAuths[0]->nonce );
270
271       if( sessionCmdRval == TPM_RC_SUCCESS )
272       {
273           // If the command was successful, check the
274           // response HMAC to make sure that the
275           // response was received correctly.
276           rval = CheckResponseHMACs( sysContext, sessionCmdRval,
277                   &nvCmdAuths, TPM20_INDEX_PASSWORD_TEST,
278                   TPM20_INDEX_PASSWORD_TEST, &nvRspAuths );
279           CheckPassed( rval );
280       }
281
282       // Check that write and read data are equal.
283       if( memcmp( (void *)&nvReadData.t.buffer[0],
284               (void *)&nvWriteData.t.buffer[0], nvReadData.t.size ) )
285       {
286           printf( "ERROR!! read data not equal to written data\n" );
287           Cleanup();
288       }
```

Cleanup: remove the NV index.

```
289
290       //
291       // Now cleanup:  undefine the NV index and delete
292       // the NV index's entity table entry.
293       //
294
295       // Setup authorization for undefining the NV index.
296       nvCmdAuths.cmdAuths[0]->sessionHandle = TPM_RS_PW;
297       nvCmdAuths.cmdAuths[0]->nonce.t.size = 0;
298       nvCmdAuths.cmdAuths[0]->hmac.t.size = 0;
299
300       // Undefine the NV index.
301       rval = Tss2_Sys_NV_UndefineSpace( sysContext,
302               TPM_RH_PLATFORM, TPM20_INDEX_PASSWORD_TEST,
303               &nvCmdAuths, 0 );
304       CheckPassed( rval );
305
306       // Delete the NV index's entry in the entity table.
307       rval = DeleteEntity( TPM20_INDEX_PASSWORD_TEST );
308       CheckPassed( rval );
309   }
```

I've demonstrated how to send single commands using an HMAC session. Now we need to consider multiple commands and how the nonces work.

Using an HMAC Session to Send Multiple Commands (Rolling Nonces)

The nonceTPM changes after every successful TPM command executed within a session. nonceCaller can be changed if the caller so desires. Because the nonces figure into the HMAC calculation, replay attacks are prevented. The HMAC calculation is as follows:

```
authHMAC := HMAC_sessionAlg ((sessionKey || authValue), (pHash || nonceNewer ||
                            nonceOlder
                             { || nonceTPMdecrypt } { || nonceTPMencrypt }
                             || sessionAttributes))
```

In this equation, notice the nonceNewer and nonceOlder parameters. On a command, nonceNewer is the nonceCaller, and nonceOlder is the last nonceTPM. For a response, nonceNewer is the current nonceTPM, and nonceOlder is the nonceCaller from the command. For now, ignore the decrypt and encrypt nonces because they're only

used for decrypt and encrypt sessions.[16] This section describes the mechanics of how the nonces are used in multiple commands in an HMAC session. A sequence of multiple commands in an HMAC session works like this (refer to Figure 13-13 and Listing 13-2):

1. When an HMAC session is started, `nonceCaller1` is sent to the TPM and `nonceTPM1` is received from the TPM. This happens in the `StartAuthSessionWithParams` call, lines 72–75 in Listing 13-2.
2. Every time a command is successfully authorized, a new `nonceTPM` is generated. This is called "rolling" the nonce. The caller can also change the `nonceCaller` before each command that is sent using the session, if desired. Look at the calls to `RollNonces` in Listing 13-2 on lines 198, 217, 245, and 269.
3. On the next session command:
    a. For the command HMAC, `nonceTPM1` is used as the `nonceOlder` parameter. `nonceCaller2`, sent with this command in the authorization area for the session, is used as `nonceNewer`.
    b. For the response HMAC, `nonceCaller2` is used as `nonceOlder`. `nonceTPM2`, sent with the response in the authorization area for the session, is used as `nonceNewer`.
4. For subsequent commands, this pattern repeats, with nonceCaller and nonceTPM flip-flopping between nonceNewer and nonceOlder in the HMAC calculation depending on whether the HMAC is being calculated on the command or response.
5. This pattern repeats until the session is closed. The nonces changing and the fact that they're used in command and response HMAC calculations prevent replay attacks.

[16]Because the `nonceTPM` figures into both the command and response HMACs, the obvious question is, what's the purpose of the `nonceCaller`? The answer (from the TPM specification writer) is that if the caller didn't trust the TPM to generate `nonceTpm` values with enough randomness, the caller could specify sufficiently random `nonceCaller` values to overcome this deficiency.

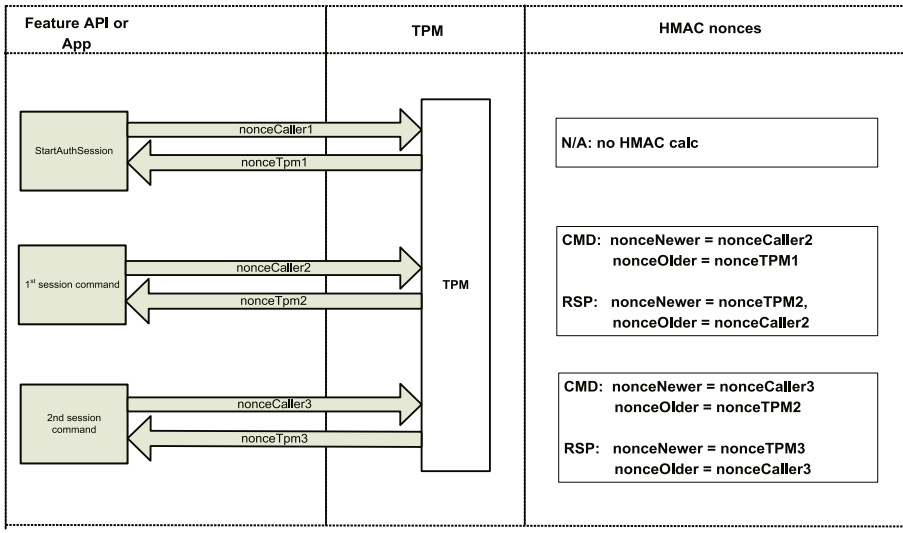

Figure 13-13. *Nonces used in an HMAC session to prevent replay attacks*

HMAC Session Security

What makes HMAC sessions secure? Basically, three aspects of HMAC sessions are used to secure commands:

- *Session key*: The bind `authValue` and `salt` are secrets that should be known only to the caller and the TPM. Both of these values are used in calculating the session key. An attacker who doesn't know these values can't calculate the session key. Because the session key is used to create the HMAC key, this means the attacker can't successfully send commands to or receive responses from the TPM. This this prevents man-in-the-middle attacks.
- *HMAC*: The session key and the entity's `authValue` are used to generate the HMAC key. The `authValue` of the entity being accessed is a secret that should only be known to the caller and the TPM. Again, this means the attacker can't successfully mount man-in-the-middle attacks.
- *Nonces*: The nonces are used to prevent replay attacks. The nonces figure into the HMAC calculation, which can't be properly performed without using the correct nonces. Since the nonces keep changing, a command byte stream can't be replayed.

As long as the secrecy of the bind `authValue`, `salt`, and entity `authValue` are maintained, attackers can't authorize actions on the entity, and the rolling nonces prevent replay of commands.

HMAC Session Data Structure

To use an HMAC authorization, the caller fills in the command authorization block as shown in Figure 13-14.

Session handle	A four-octet value indicating the session number associated with this data block.
Size field	A two-octet value indicating the number of octets in *nonce*.
Nonce (from the caller)	If present, an octet array that contains a number chosen by the caller.
Session attributes	A single octet with bit fields that indicate session usage.
Size field = `sizeof` HMAC	A two-octet value indicating the number of octets in *authorization*.
Authorization = HMAC	If present, an octet array that contains an HMAC. This is the HMAC generated by the code sending the command. This HMAC is calculated on the parameters being sent to the TPM. The TPM independently calculates this HMAC and compares it to the one sent down the wire to verify that the command byte stream wasn't corrupted.

Figure 13-14. *Command HMAC authorization area*

The code that fills in the command authorization blocks is in Listing 13-2 on lines 19–21, 150, 189–195, 198 (sets the `tpmNonce`), and 202–205 (sets the HMAC in the authorization area).

The response authorization area looks like Figure 13-15.

Size field	A two-octet value indicating the number of octets in *nonce*.
Nonce (from the TPM)	If present, an octet array that contains a number chosen by the TPM.
Session attributes	A single octet with bit fields that indicate session usage.
Size field	A two-octet value indicating the number of octets in *acknowledgment*.
Acknowledgment = HMAC	If present, an octet array that contains an HMAC. This HMAC is generated by the TPM over the response data returned by the command. The caller receives the response, independently calculates the HMAC of the response, and verifies it against the TPM-generated HMAC to verify that no data corruption has occurred.

Figure 13-15. *Response HMAC authorization area*

The code that sets up the response authorization blocks is in Listing 13-2 on lines 24–26. The call to the one-call function returns the authorization area from the TPM in `nvRspAuths`, and the call to `CheckResponseHMACs` on lines 224–226 verifies that the HMAC in the response authorization is correct.

This concludes the deep dive into HMAC sessions. Now the water gets even deeper with a discussion of the most feature-rich and complicated authorizations: policy or extended authorization.

Policy Authorization

Policy authorization, also known as Extended Authorization (EA), is the Swiss army knife of TPM authorizations. With the right expertise, you can do just about any kind of authorization with it. This section and the following chapter, Chapter 14, aim to give you the knowledge required to use the incredible power of EA. In this section, we describe how EA works at a high level, the high-level policy authorization lifetime, and each of the steps in that lifetime: policy hash creation, entity creation or alteration, policy session creation, and policy session use. We will also explore the security properties of EA. As much as possible, this section doesn't describe individual policy commands; the next chapter describes those in detail.

How Does EA Work?

At a high level, EA enables very expressive policies. EA, like HMAC and password authorizations, is used to authorize actions on a TPM entity. Some examples of the controls that can be enforced before authorizing an action are:

- Requiring certain values in a specified set of PCR registers
- Requiring a certain locality
- Requiring a certain value or range of values in an NV index
- Requiring a password
- Requiring physical presence
- Requiring sequences of conditions

And there are many more. These can be combined in `AND` and `OR` combinations that result in an infinite number of policy variations. Policy authorizations allow considerable complexity and creativity in authorizations. Policy authorizations are the "mother of all complex authorizations."

For a command to be authorized by a policy authorization, two things must be correct:

- Each policy command "asserts" that some condition(s) are true. If the specified conditions for each policy command aren't true, then the authorization fails. This failure can happen either:
  - *At the time of the policy command*: This is an *immediate* assertion, and this failure occurs before ever getting to the command to be authorized. This failure means the `policyDigest` for the session isn't hash-extended by the policy command. If the policy command is successful, the `policyDigest` is hash-extended with the proper values.
  - *At the time of the command being authorized*: This is a *deferred* assertion. In this case, the `policyDigest` is hash-extended with data that indicates that the particular policy command was executed. Testing of the conditions is deferred until the time of the action being authorized.

■ **Note** Some commands can be combined assertions, which means both immediate and deferred conditions must be valid for the assertion to pass.

- At authorization time:
  - Any deferred conditions are checked. If any of these fail, the command isn't authorized.
  - The entity's `authPolicy` is compared to the policy session's `policyDigest`. If they're equal, the command is authorized, and it executes. If they aren't equal, the authorization fails. Basically, if the `authPolicy` is equal to the `policyDigest`, this is proof that the required policy commands were executed, any immediate assertions generated by those commands passed, and that all this occurred in the correct sequence before the command being authorized.

Now you've seen two time-related terms: *policy command time* and *authorization time*. All the time intervals related to policy authorizations need to be defined precisely in order for you to understand policy authorizations.

Policy Authorization Time Intervals

In working with policy authorizations, four distinct time intervals must be considered. These are implied by various sections of the specification but not specifically delineated:

- *Build Entity Policy time*: The time interval when the `authPolicy` used to create an entity is built. There are two ways to create this `authPolicy`:
  - Software can replicate the policy calculations done by the TPM.
  - A trial policy session can be created. The policy commands used to generate the `policyDigest` are sent to the TPM. During a trial policy session, all assertions pass; the purpose of the trial policy session is to generate the `policyDigest` as if all the assertions passed. After all the policy commands are sent to the TPM, the `policyDigest` can be read from the TPM using the `TPM2_GetPolicyDigest` command.

■ **Note** Policies may be, and often are, reused for creating multiple entities and for authorizing many actions.

- *Create Entity time*: The time when the entity is created. If the entity will use an `authPolicy`, the policy digest created at Build Entity Policy time is used to create the entity.

■ **Note** Because an entity's name is created at Create Entity time, the policy digest input when creating the entity (for example, `authDigest`) can't include the entity's name.

- *Build Policy Digest time*: After a policy session is started, during this time interval, policy commands are sent to the TPM in preparation for a command to be authorized. These commands cause the session's `policyDigest`, which is maintained inside the TPM, to be hash-extended with policy command-specific values.
- *Authorization time*: The time when the command to be authorized is sent to the TPM. At this time the session's `policyDigest` must match the entity's `authPolicy`, and any deferred assertions must pass.

To summarize, a policy calculation is usually performed twice—once at Build Entity Policy time and once at Build Policy Digest time:[17]

- The first time is to create a policy hash used as the `authPolicy` when creating an entity.
- The second time occurs before authorizing an action on an entity: a policy hash is built up in the session context inside the TPM. When the command to be authorized is executed, the session's policy hash is compared to the `authPolicy` of the entity. If they match, the authorization is successful.

All policy commands do two or three things, and they do these things at the following time intervals:

- They check a condition or conditions (the *assertion*). This is done at Build Policy Digest time (immediate assertion) or Authorization time (deferred assertion), or some combination of the two (combined assertion).
- They hash-extend the current session policy digest with policy command-specific data. This is done at Build Entity Policy time and Build Policy Digest time.
- They *may* update other session state. This session state is used for deferred or combined assertions to indicate what deferred conditions should be tested at authorization time. These updates are done at Build Policy Digest time.

Now that you understand the various time intervals, let's look at a typical policy session lifetime.

Policy Authorization Lifecycle

The typical steps in a policy authorization lifecycle are very similar, with some additions, to the lifecycle steps used for password and HMAC sessions:

1. Build the entity policy.
2. Create the entity using the policy digest created in step 1.

[17]It should be noted that in some cases, a single real policy session can be used to generate the policy for both the creation of the entity and authorizing actions within the session; in this case, the policy digest is calculated only once. For instance, the following sequence would work: start a real policy session, send the TPM2_PolicyLocality command, get the policy digest, create the entity using the policy digest, and authorize a command using the policy session. This reverses the usual order of creating the entity and then starting the real policy session. It probably isn't very useful for most normal uses, but an understanding of this provides insight into how policy sessions operate. This only works for cases where the policy assertions can be can be satisfied before the entity is created.

■ **Note** Steps 1 and 2 are typically performed long before the remaining steps. And the remaining steps can occur multiple times to authorize multiple actions on the entity.

3. Start a policy session.
4. Using the policy session, send policy commands to fulfill the required authorization.
5. Perform the action on the entity that requires authorization.

Let's look at each of these steps in detail, with the applicable line numbers from Listing 13-2. For brevity's sake, line numbers are listed only for code that is unique to policy sessions.

Building the Entity's Policy Digest

The first task in using a policy session is to determine what the authorization policy will be: for example, what entities need to be protected, what actions on those entities need to be restricted, and the exact nature of those restrictions. Then, the policy digest must be created; this step corresponds to the Build Entity Policy time interval described previously. There are two ways to create a policy digest: use a trial policy session, or create the policy digest with software that emulates the actions of the TPM in creating a policy digest. I will describe both of these using a simple example; the code uses a trial policy session.

An example policy might allow an NV index of 0x01400001 to be written or read by someone who knows its `authValue`. In this case, building the entity policy using a trial policy session can be done as follows:

1. Start a trial policy session using the `TPM2_StartAuthSession` command. The main inputs of concern for a policy session are:
   a. `sessionType = TPM_SE_TRIAL`. This is what configures the session as a trial policy session.
   b. `authHash = TPM_ALG_SHA256`. This sets the hashing algorithm used for generating the `policyDigest`. I chose SHA256, but any hashing algorithm supported by the TPM can be used here.

   This command returns a policy session handle, call it H_{ps}. Lines 63, 66, and 72–75 start the trial policy session.
2. Send a `TPM2_PolicyAuthValue` command with the following inputs (see lines 78–79): `policySession =` H_{ps}.

   This command extends the session's policy digest as follows:

$$policyDigest_{new} := H_{policyAlg}(policyDigest_{old} \,||\, TPM\_CC\_PolicyAuthValue)$$

where policyAlg is the hash algorithm set by the TPM2_StartAuthSession command, and $policyDigest_{old}$ is the buffer of length equal to the size of the policy digest that corresponds to the policyAlg with all bytes set to 0.

3. Send a TPM2_GetPolicyDigest command (lines 83–85). This command returns the policy digest, $digest_{ps}$.

Alternatively, to calculate $digest_{ps}$ in software, the software needs to duplicate the policy digest calculation in step 2. Appropriate calls to a crypto library such as OpenSSL can be used to accomplish this.

Once the policyDigest has been calculated or created, the NV index can be created to use the policyDigest for authorization of write operations to the index. Unlike a password or HMAC authorization, after the NV index is created the policyDigest used to access an NV index or any other entity can't be directly changed. There are advanced policy commands that can accomplish this through a policy-specific method of indirection, but that topic is described in the next chapter.

Creating the Entity to Use the Policy Digest

Now we need to create the index in such a way as to allow writes with the policy authorization; this step corresponds to the Create Entity time interval described previously. This is done by sending a TPM2_NV_DefineSpace command with the following inputs (this is done by the call to the DefineNvIndex function):

- auth = TPM2B that contains the authValue used to access this NV index (lines 42-44).
- publicInfo.t.nvPublic.nvIndex = 0x01400001 (lines 115-117).
- publicInfo.t.nvPublic.nameAlg = TPM_ALG_SHA256. This is the hash algorithm used to calculate the index's name, and this algorithm must be the same as the policyAlg used to calculate the policyDigest, whether this was done by a trial session or by software. See lines 115-117.
- publicInfo.t.nvPublic.attributes.TPMA_NV_POLICYWRITE = 1 and publicInfo.t.nvPublic.attributes.TPMA_NV_POLICYREAD = 1. This configures the index to allow reads and writes only if the policy is satisfied. See lines 109–110.
- publicInfo.t.nvPublic.authPolicy = the TPM2B that contains the policyDigest, $digest_{ps}$. See lines 83-85 and 115-117.
- publicInfo.t.nvPublic.dataSize = 32. This indicates the size of the data contained in the NV index; in this case, the index is configured to be only 32 bytes wide. See lines 115-117.
- Set the NV index's auth value. See lines 42–44.

This command creates an NV index that can only be written if the policy is satisfied. The next step is to create a real—that is, non-trial—policy session and use it to authorize writes to the NV index.

Starting the Real Policy Session

Start a real policy session using the `TPM2_StartAuthSession` command. The main inputs of concern for a policy session are as follows (see lines 152 and 154-156):

- `tpmKey = TPM_RH_NULL`
- `bind = TPM_RH_NULL`

■ **Note** The `tpmKey` and `bind` settings mean this is an unbound and unsalted session. These settings were chosen in order to keep this example as simple as possible; they're also the most common way that policy sessions are used. The goal here is to understand the process and avoid low-level details as much as possible.

- `sessionType = TPM_SE_POLICY`. This is what configures the session as a real—that is, non-trial—policy session.
- `authHash = TPM_ALG_SHA256`. This sets the hashing algorithm used for generating the `policyDigest`. Because we used SHA256 when creating the `policyDigest`, we must use this same algorithm when starting the real policy session.

This command returns a policy session handle, H_{ps}. Now we can use this policy session to send commands to authorize actions on the NV index.

Sending Policy Commands to Fulfill the Policy

Using the policy session created in the previous step, the code now sends the same sequence of policy commands that it used to create the NV index's `policyDigest` at Build Entity Policy time; this step corresponds to the Build Policy Digest time interval described previously. In this case, the sequence is very simple, and we only need to send one policy command, a `TPM2_PolicyAuthValue` command with the following input (lines 177–179): `policySession =` H_{ps}.

In response to this command, the TPM does two things:

- It extends the policy session's policy digest just as it did for the trial session at Build Entity Policy time.
- Because `TPM2_PolicyAuthValue` is a deferred assertion, it saves some state information into the policy session's context so that it knows to check the HMAC at Authorization time.

At this point, the policy authorization is completely "locked and loaded" to authorize the action. The next thing that happens is that we attempt to write to the NV index.

Performing the Action That Requires Authorization

This step corresponds to the Authorization time interval described earlier. We write to the NV index, and, if it's been authorized correctly, the write completes successfully. To do this step, send the `TPM2_NV_Write` command with the following inputs:

- `authHandle = 0x01400001`. This handle indicates the source of the authorization. See lines 211-214.
- `nvIndex = 0x01400001`. This is the NV index to be authorized. See lines 211-214.
- The authorization area for `authHandle` must have the following settings:
  - `authHandle` = the policy session handle, H_{ps}. See lines 189-190.
  - `nonceCaller` = whatever nonce the caller wants to use. This can even be a zero-sized nonce. See lines 191-192.
  - `sessionAttributes = 0`. See lines 193-194.
  - `hmac.t.buffer` is set to the HMAC of the command. The HMAC key is the session key concatenated with the `authValue` of the NV index. See lines 202-205.
- `data` = 0xa5. See lines 166-169 and 211-214.
- `offset = 0`. See lines 211-214.

In response to this command, the TPM checks that `policySession->policyDigest` matches the `authPolicy` of the entity being accessed. Then it checks that the HMAC is correct. If both checks pass, the write proceeds and the command completes successfully.

■ **Note** You may have noticed that in Listing 13-2, because the policy case uses a `TPM2_PolicyAuthValue` command, the HMAC and policy cases are very similar. The main difference is that the policy case requires more work. The obvious question is, if a policy session that uses `TPM2_PolicyAuthValue` requires more work, why wouldn't we just use an HMAC session? The answer, which is expanded in the next chapter, is that a policy session allows many other factors besides the authorization value to be combined, creating a much more configurable and, possibly, secure authorization.

You've now seen a complete policy authorization from cradle to grave. This was a very simple example, but it should form a good basis for understanding the more complex policy authorizations in the next chapter.

To finish this chapter, we unify the lifecycles for password, HMAC, and policy authorizations into one single lifecycle.

Combined Authorization Lifecycle

The typical steps in an authorization lifecycle are the following:

1. For HMAC or policy sessions, an `authValue` or `authPolicy` must be determined before creating the entity:
    a. If actions on the entity will be authorized using a policy session, precalculate the `authPolicy` policy hash.
    b. If actions on the entity will be authorized using a password or HMAC session, determine what the shared secret will be.
2. Create the entity to be accessed using an authorization value (`authValue`) and/or policy hash (`authPolicy`), or change the `authValue` value for an existing entity (changing the `authPolicy` for an entity is done by a different means and is described in the next chapter):
    a. The entity's `authValue` will be used for either password authorizations or HMAC authorizations. For password authorizations, the `authValue` will be used as a clear-text password. For HMAC authorizations, the `authValue` will be used to generate session HMACs.
    b. The entity's `authPolicy` is used to determine if the proper policy assertions have passed before the command to be authorized. This policy hash must be precalculated before creating the entity; hence step 1a.
3. Calculate the HMAC. For policy sessions that don't use an HMAC, this step can be skipped.
4. In the case of an HMAC or policy authorization, start the HMAC or policy session.
5. Do an authorized action using the authorization. The authorization passes if:
    a. The password sent during the command matches the entity's `authValue`.
    b. The HMAC sent during the command matches the HMAC calculated by the TPM. Both of these HMACs are derived, in part, from the `authValue` of the entity.

c. The `policyDigest` of the policy session at Authorization Time matches the `authPolicy` of the entity. This policy hash derives from a variety of factors determined by the policy command(s) used to create the `policyDigest`. Also, any deferred assertions must pass for the authorization to be successful.

6. In the case of an HMAC session, calculate the expected response HMAC, and verify it against the one returned by the TPM.

These steps are represented in relative time order, but many other actions could occur between them. Also, a single `policyDigest` can be used to authorize multiple actions to multiple entities. Similarly, a single HMAC session can be used to authorize multiple actions to multiple entities. The exact mechanics of these steps vary with the authorization type, and these differences were described previously, but each of these steps must be performed for all authorizations with the following exceptions:

- Steps 3–4 aren't required for password authorizations.
- Step 6 isn't required for password or policy authorizations.

For more code examples of policy sessions, see the `TestPolicy` function in the TSS SAPI test code.

Summary

This concludes the discussion of authorizations and sessions. Congratulations on making it this far! If you understand this chapter, you're well on your way to becoming a TPM 2.0 master.

This chapter described the general concepts of authorizations and sessions and tried to clarify their differences. You looked at the command and response authorization areas, and lifecycles for password, HMAC, and policy authorizations. Then you saw an overall authorization lifecycle.

This may have felt like drinking from a fire hose, and that's because it was! This is one of the most difficult areas of the TPM to understand; good comprehension of this material will aid you immeasurably in understanding and using TPM 2.0 devices. The next chapter describes the most powerful of authorizations—policy authorizations—in detail, with a description of each of the policy authorization commands and use cases for them.